Week 5

Primitives and References

Classes and Objects

Introduction to OO Design & UML

Creating your own classes

Methods

Instance variables

Data encapsulation

Constructors

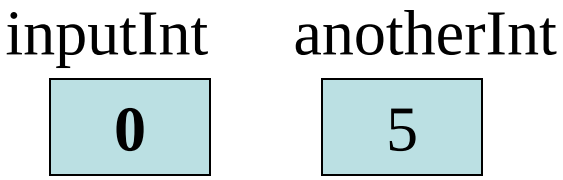
StringTokenizer class

Introduction to Regular Expressions `

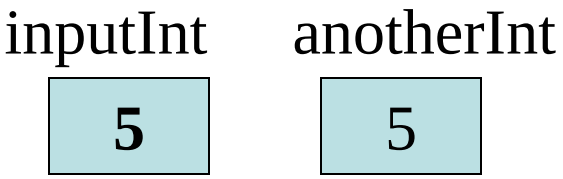
Read Pages 130-153
And
Pages 214-223

Assignment Primitives

```
int inputInt = 0;
int anotherInt = 5;
```

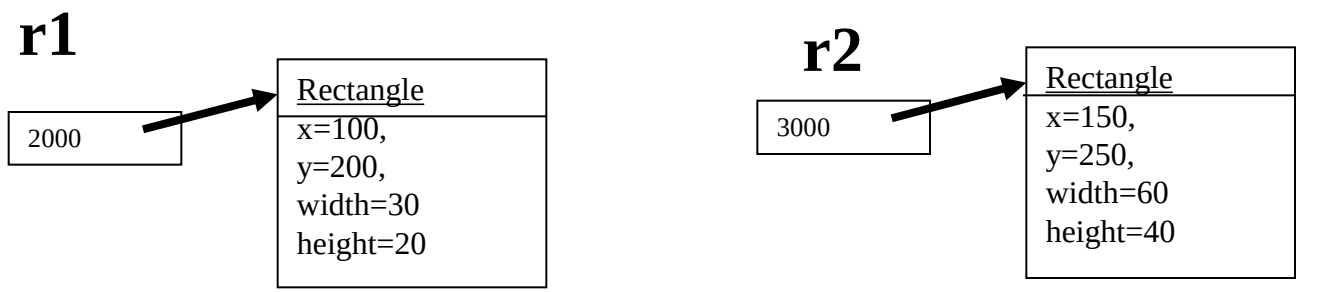



```
...
inputInt = anotherInt;
```

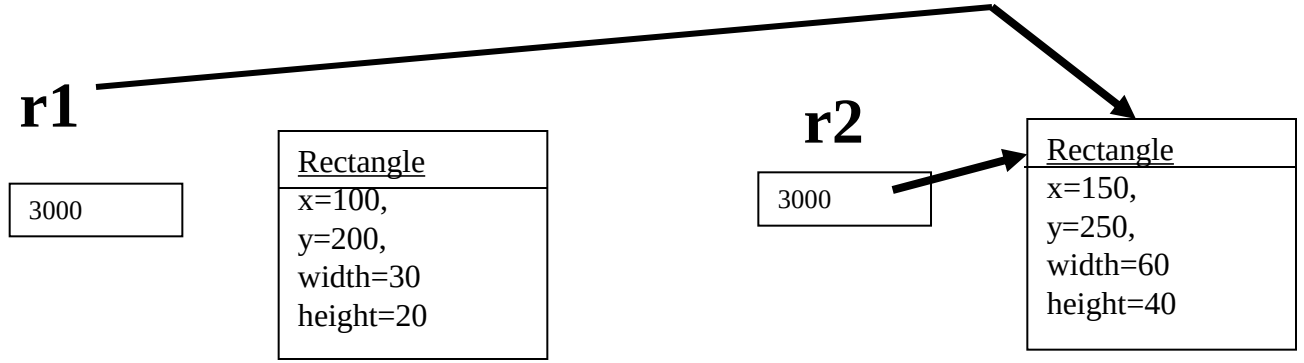


Assignment References

```
Rectangle r1 = new Rectangle(100,200,30,20);
Rectangle r2 = new Rectangle(150,250,60,40);
```




```
r1 = r2;
```



Sample program

```
import java.awt.*;
public class RectangleObjects {
  public static void main (String[] args) {
    Rectangle r1 = new Rectangle(100,200,30,20);
    Rectangle r2 = new Rectangle(150,250,60,40);
    r1 = r2;
    r1.translate(100,50);
     System.out.println("r1: x= "+r1.getX()+" y= "+r1.getY()+
                        " w="+r1.getWidth()+" w="+r1.getHeight() );
     System.out.println("r2: x= "+r2.getX()+" y= "+r2.getY()+
                        " w="+r2.getWidth()+" w="+r2.getHeight() );
  }
}
```

output

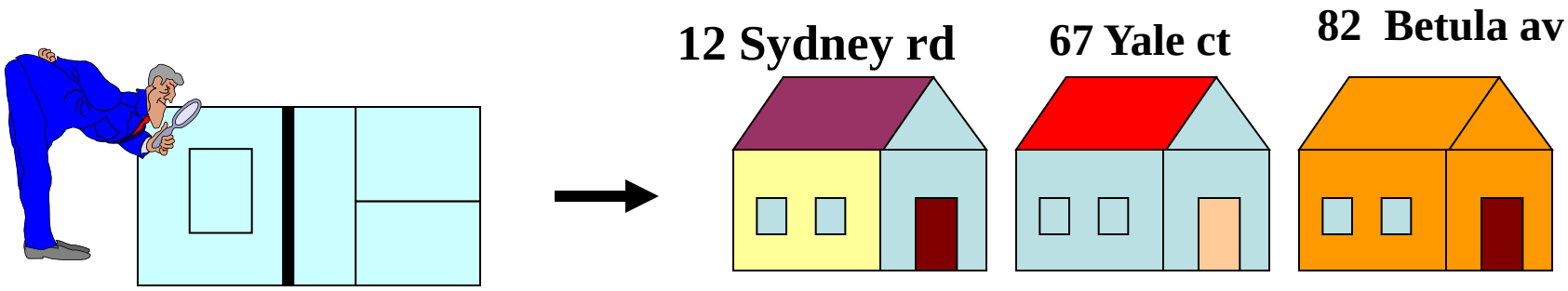
r1: x=250.0 y = 300.0 w = 60.0 h = 40.0
r2: x=250.0 y = 300.0 w = 60.0 h = 40.0

What will be the output if we replace r1 = r2; with:

(a) r2 = r1;

(b) r1.setBounds(r2);

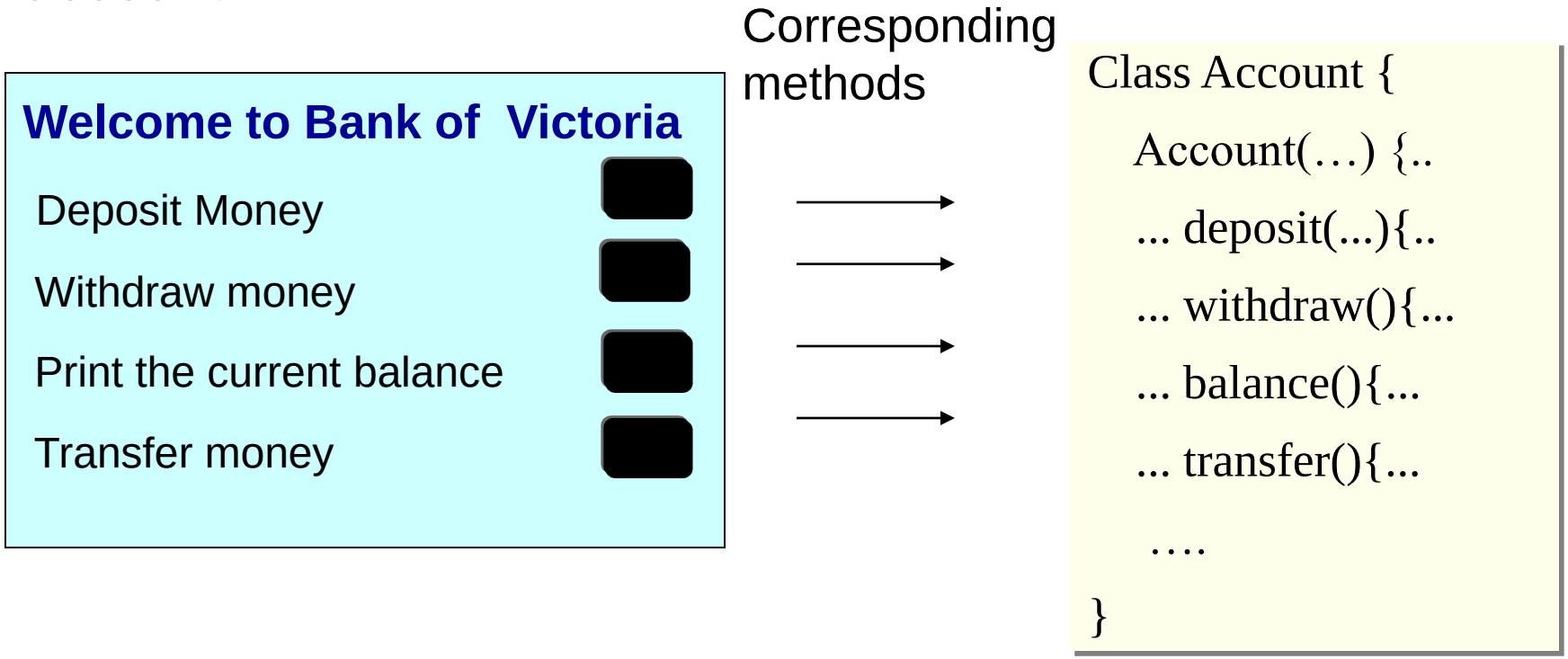
Classes and Objects



- We say that an object is an instance of a class.
- A class is a template, a model from which instances can be created
- We have created many instances of String class in the previous weeks.
- To model parts, we first create a Part class and then instantiate many Part objects (instances)

Deciding Object Behavior for our own classes

- Suppose we want to create a class to model a bank account.
- What kind of operations can you carry out with a bank account ?

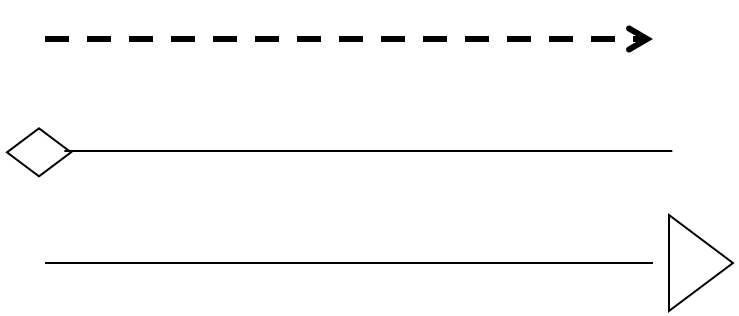


Common Relationship between classes

- Dependency
  - A class depends on another class if it manipulates objects of another class
- Aggregation
  - Takes place when objects of one class contain objects of another class
- Inheritance
  - An inherited class exhibits additional behaviors

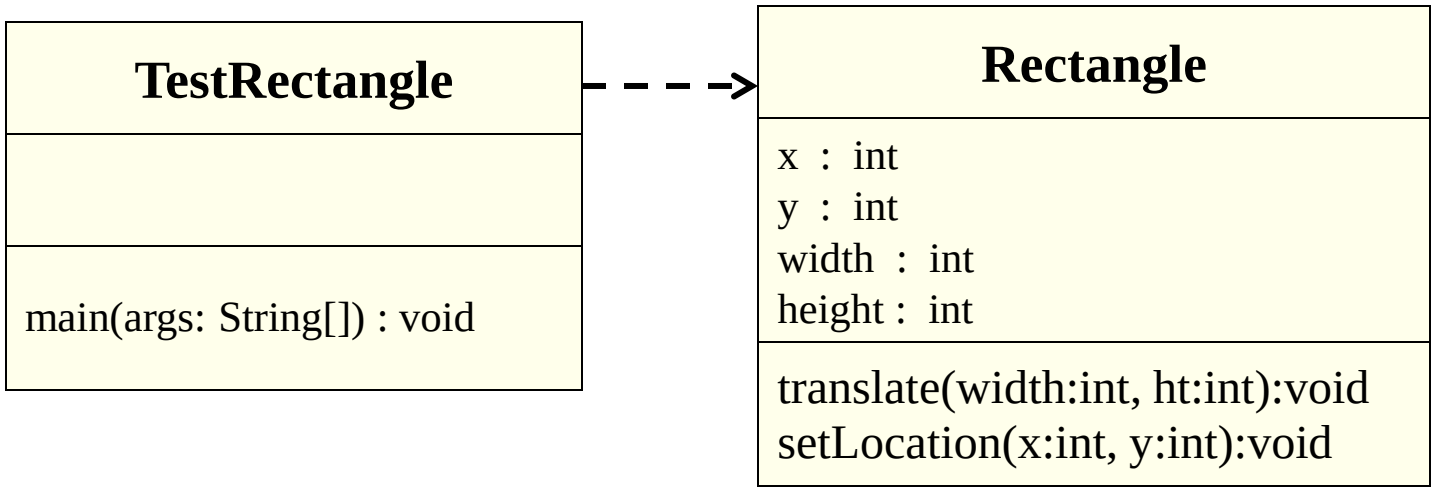
UML Connectors for classes

- Dependency
- Aggregation
- Inheritance



UML Class Diagrams

- In a UML class diagram each class is represented as a rectangle. UML diagrams are versatile – we can include whatever details are needed.
- It can show up to three parts
  - Class name
  - Attributes
  - Methods
- Relationship between classes can be shown using appropriate arrows



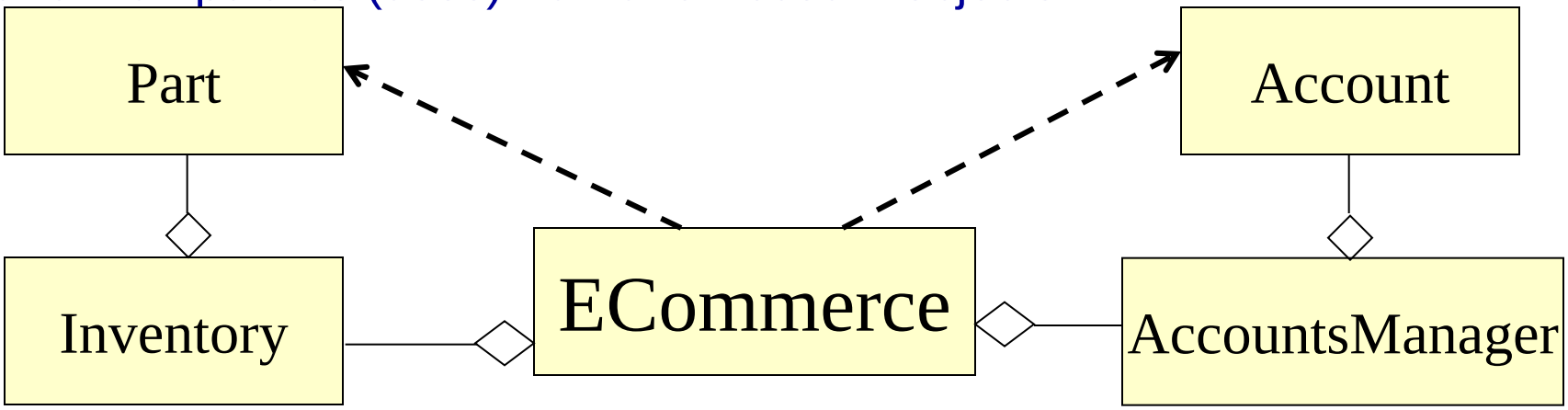
Getting the Big Picture

(Final Assignment 2001)

Suppose we are asked to write an E-commerce system.

We may have Identified 5 main classes, namely

- Part class - to model parts bins (getQty(), getPrice())
- Inventory Object contain all the Part objects
- Account class - to model bank accounts (withdraw(), getBalance() …)
- AccountsManager object contain all Account objects
- ECommerce object contains AccountsManager and Inventory objects and manipulates (uses) Part and Account objects

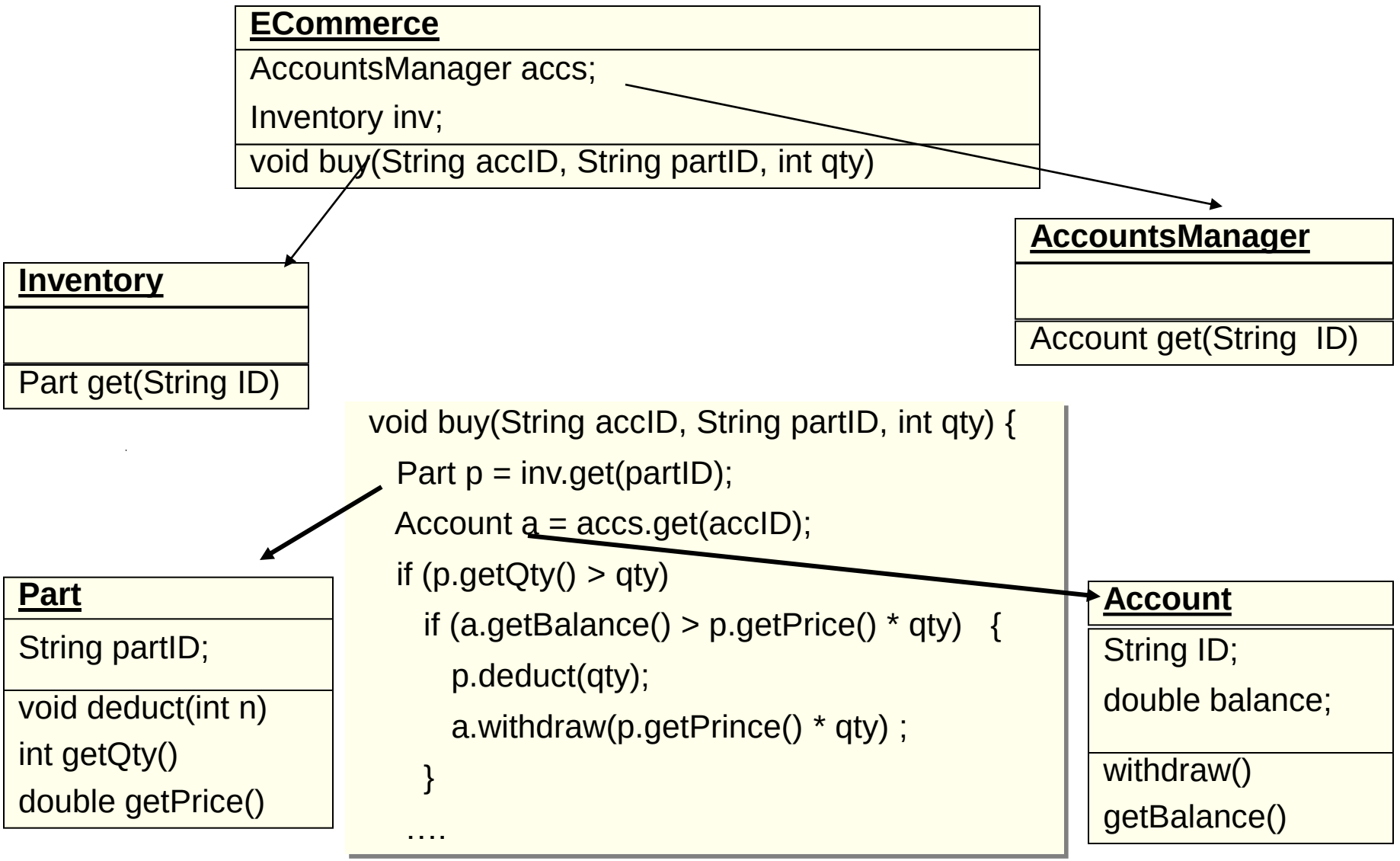


A buy operation of ECommerce class

When a customer wants to buy the buy operation Ecommerce is called passing Part and Customer ID. This operation involves:

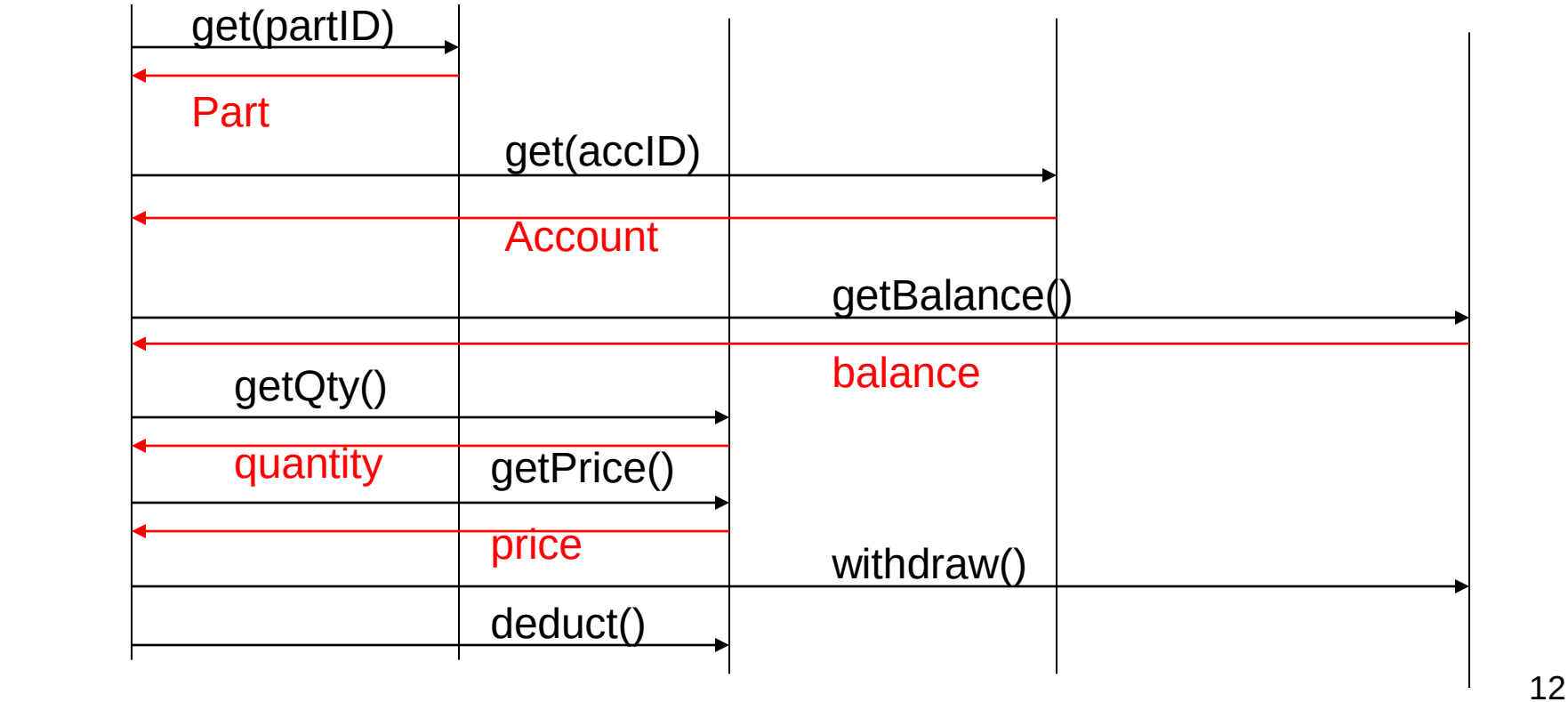
- identify the specific part object
- check whether the required quantity is available
- identify the account object
- check whether the account has sufficient funds available.
- A possible buy operation of ECommerce class is shown next.

Objects Communicate by sending messages



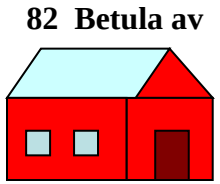
Sequence Diagram
Ecommerce: buy() method

UML sequence diagram helps understand complex control flow.



Constructing an Account object

Bill requested the builder to construct a Rochester (type) at 82 Betula avenue with red brick and blue tiles.

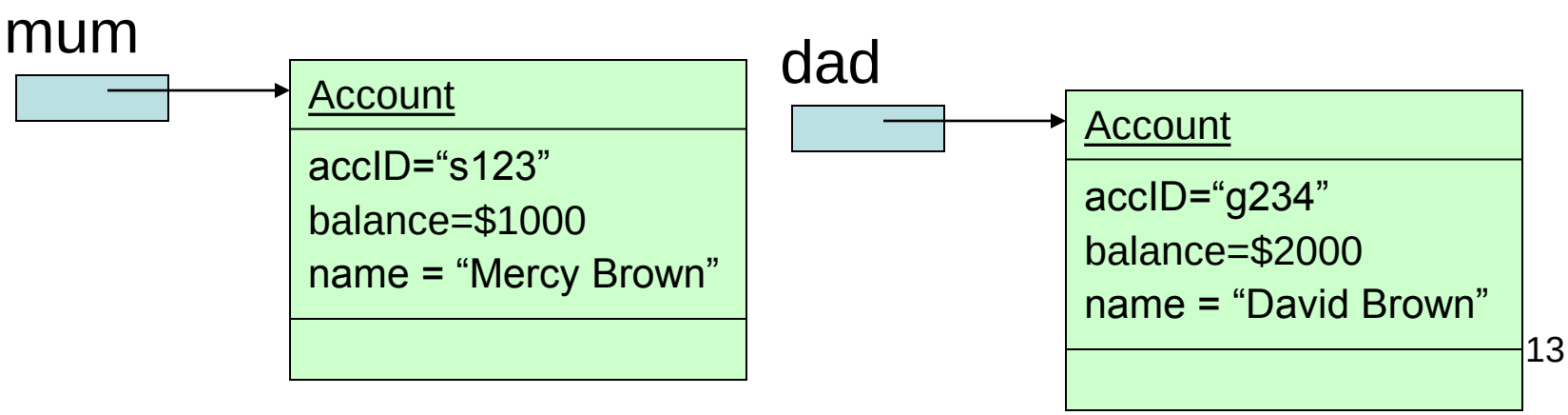


We can use Account constructor to create an Account objects specifying the arguments as in

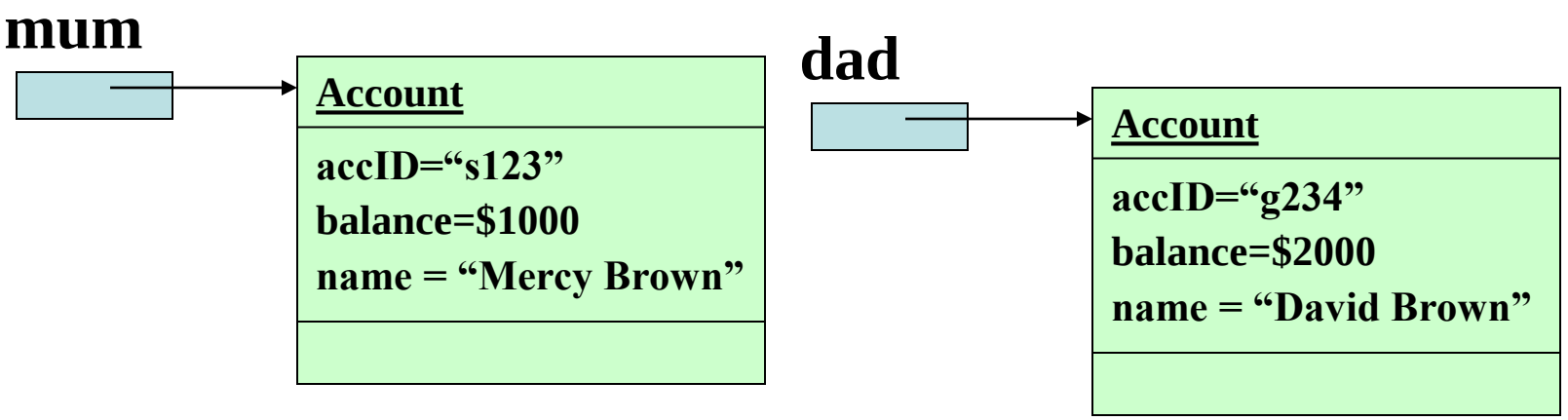
```
Account mum = new Account("s123","Mercy Brown",1000);

Account dad = new Account("g234","David Brown",2000);
```

Account constructor

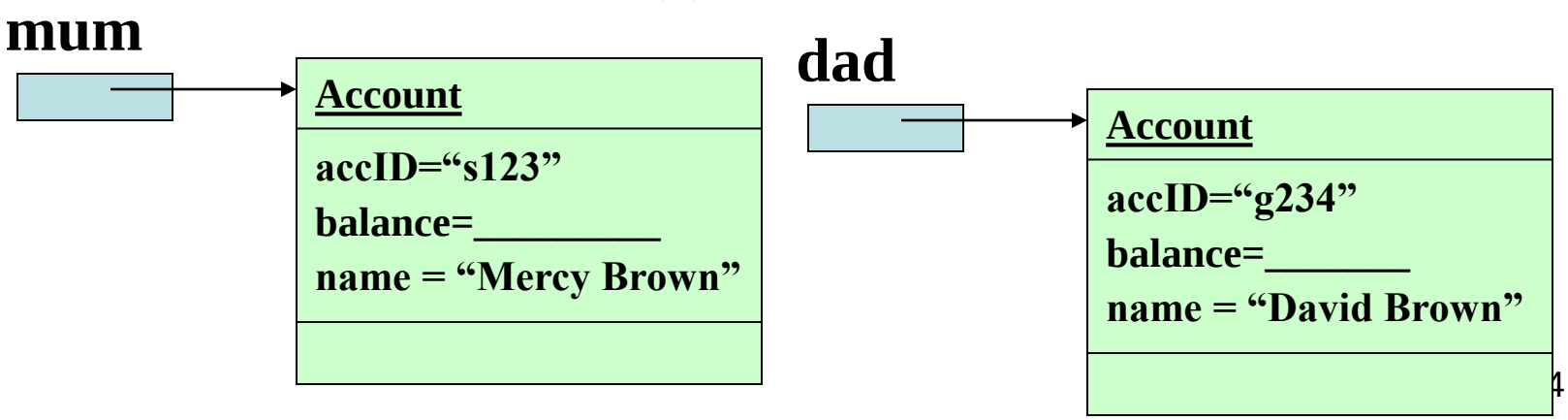
Arguments(relevant information)



What operations can we do with them ?



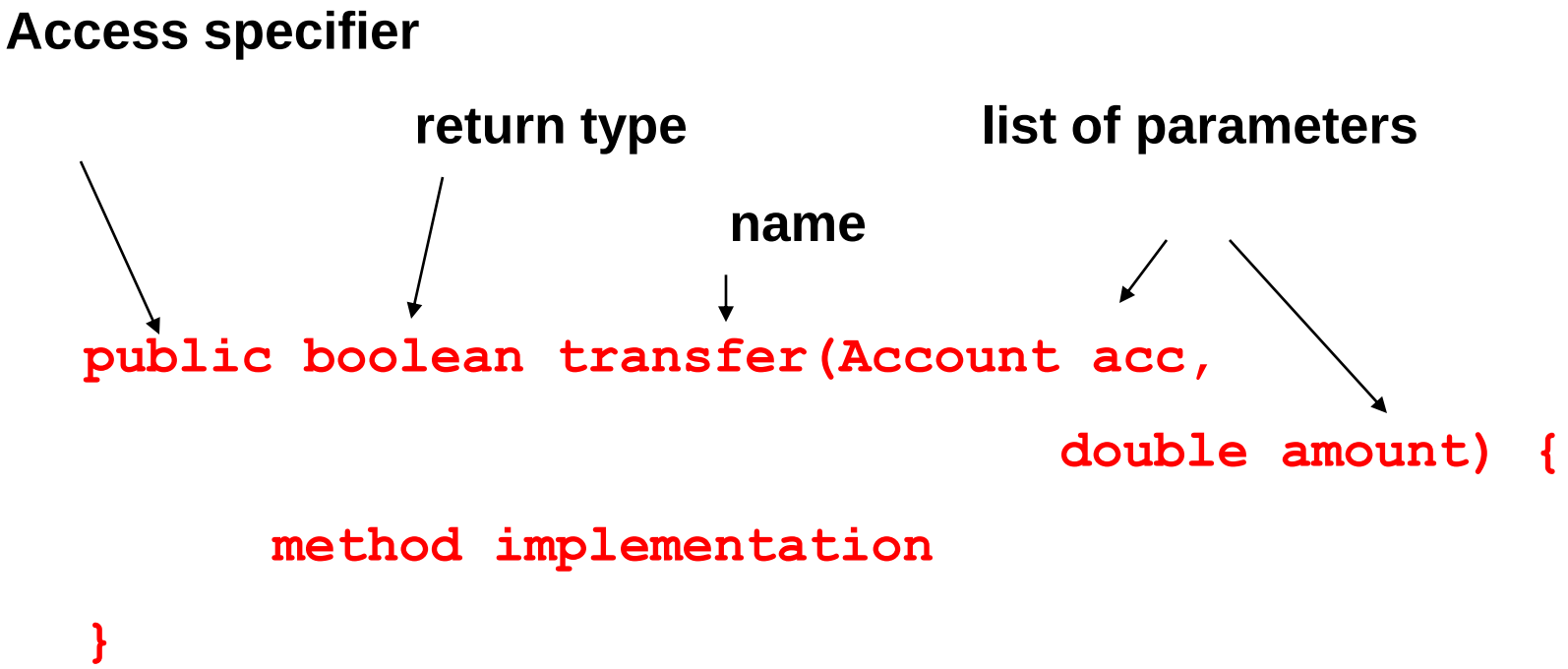

```
mum.withdraw(100);        // mum's withdrawing $100
dad.deposit(150);         // dad's depositing $150
dad.transfer(mum,500);    // dad's transferring $500
                          // to mum
```



These operations need methods

```
public void deposit(double amount) {
	method implementation
}
public boolean withdraw(double amount) {
	method implementation
}
public boolean transfer(Account acc, double amount) {
	method implementation
}
public double getBalance() {
	method implementation
}
public double getID() {
	method implementation
}
```

Parts of a methods



What makes one Account object different from another ?

Each instance stores the values reflecting its current state (such as balance, ID , name)

From the operations we have identified we should have at least three instance variables.

```
public class BankAccount
{   …
    private String name;
    private double balance;
    private String accID;
}
```

private

(not accessible by class users)

Accessing balance ...

```
Account mum = new Account("s123",
                          "Mercy Brown",1000);
// Error balance is private
double d = mum.balance;
```

But you can call the public getBalance() method to inquire about the balance:

```
// OK
double d = mum.getBalance();
```

Why private ?

Suppose we created a class to model a switch which has two possible states on and off as in:

```
class switch
{   switch() {state = 0; }
    public setOff() { state = 0;}
    public setOn() {state = 1; }
    public int state;
}
```

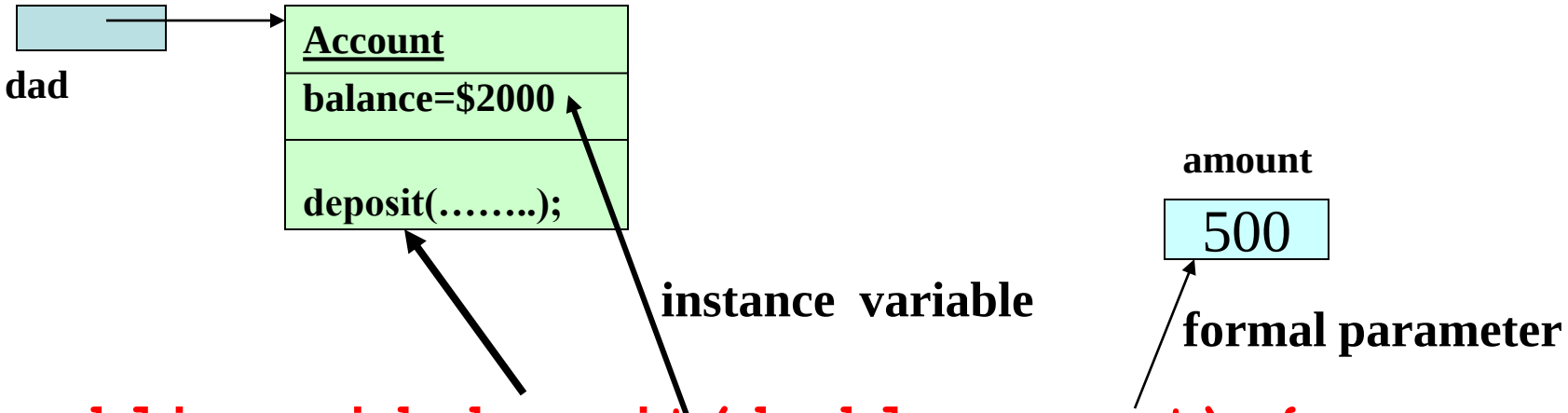
```
int on = 1;

switch a = new switch();

a.state = on;

...
```

- Can we now change state to a boolean variable (efficient) ?
- We cannot because class users have assumed it is an int and written code.
- So data should always be hidden (private) - Encapsulated.

Implementing Methods

To deposit $150 to dad we used:

```
dad.deposit(500); //deposit() of dad called
```




```
public void deposit(double amount) {
     balance = balance + amount;
}
```

Formal parameters are used for passing arguments.
They come into existence only when method is invoked.

Implementing Methods

To withdraw $100 from mum we used:

```
mum.withdraw(100); // withdraw() of mum called
```

withdraw() reduces the balance and to return true if funds are sufficient and false otherwise.

```
public boolean withdraw(double amount){
    boolean result = false;

    if ( balance > amount ) {
            result = true;
            balance -= amount;
    }
    return result;
}
```

local variable cannot be used outside the method

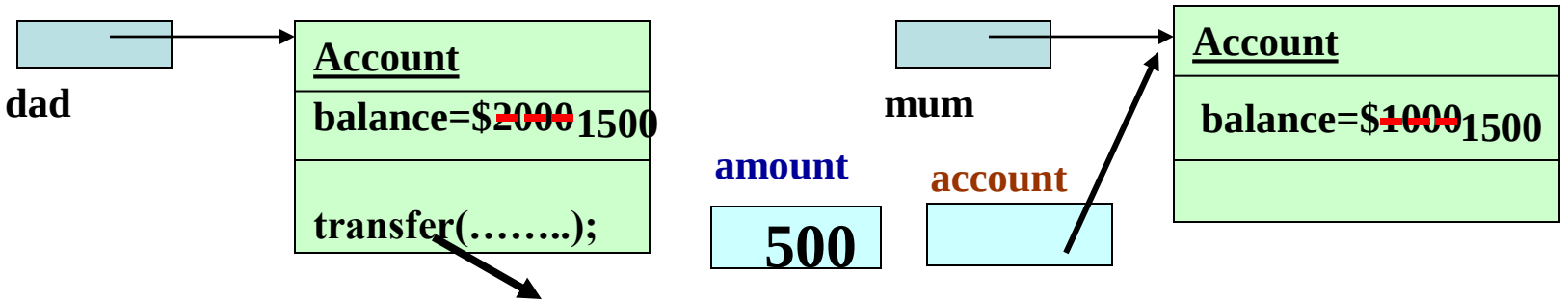
formal parameter is used for passing arguments

Local variables are used within a method. Like formal parameters they come into existence only when that method is invoked.

Implementing Methods

To deduct money from dad's balance and to credited to mum's use the statement below. It fisrt verifies dad has sufficient funds.

```
dad.transfer(mum,500); //transfer() of dad called
```




```
public boolean transfer(Account account,
                        double amount) {
   if (balance >= amount) {
      balance = balance - amount;
      account.balance += amount;
      return true;
   }
   else return false;
}
```

Accessor & Mutator methods

A method that accesses an object and returns some information about it, without changing the object is called an *accessor* method.

```
public double getBalance() {
   return balance;
}
public String getID() {
   return accID;
}
public String getName() {
   return name;
}
```

Methods that modify the state of an object are called *mutator* methods. Example deposit(), withdraw().

Constructors

- **Initializes the instance variable of an object.**
- **Has the same name as the class generally public.**
- **But unlike other methods have no return types.**
- **If no constructor provided Java will provide one with all instance variables set to default values.**
- **Account constructor is invoked twice below**

```
Account mum = new Account("s123","Mercy Brown",1000);
Account dad = new Account("g234","David Brown",2000);
```

```
public Account(String accountID,
      String accountName, double amount)
{   accID = accountID;
    name = accountName;
    balance = amount;
}
```

More than 1 constructor per class allowed?

- Yes provided the method signatures are different.
- The one below only takes two arguments. As no value is passed for balance it is set to 0.
- When called with 2 string arguments as shown below this constructor will be invoked.

```
Account poor = new Account("s123", "Mercy Brown");
```

```
public Account(String accountID,
                           String accountName){
      accID = accountID;
      name = accountName;
      balance = 0.0;
}
```

Testing the Account class

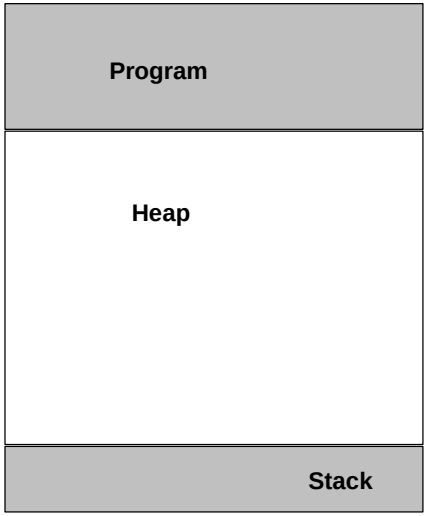
The TestAccount class in a separate file (TestAccount.java) creates two Account objects and performs the operations discussed above.

The final balance for mum and dad should be $1400 and dad $1,650 respectively.

```
public class TestAccount
{ public static void main(String args[])
  {
     Account mum = new Account("s123","Mercy Brown",1000.0);
     Account dad = new Account("g234","David Brown",2000.0);
     mum.withdraw(100);
     dad.deposit(150);
     dad.transfer(mum,500);
     System.out.println("mum bal = "+mum.getBalance());
     System.out.println("dad bal = "+dad.getBalance());
  }
}
```

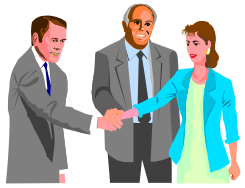
Stack and Heap Memory

- Objects created (with new) are placed in the heap.
- When there are no references to an object the garbage collector will free the space.
- Local variables and formal parameters are placed in the stack. Suppose method A() calls method B() which calls method C(). All the local variables and the formal parameters of these methods will go an the stack. When it is complete the memory space will be freed in the last in first out order.



Another class

A class to model employees to keep track of their names, salaries and their immediate superiors (bosses).



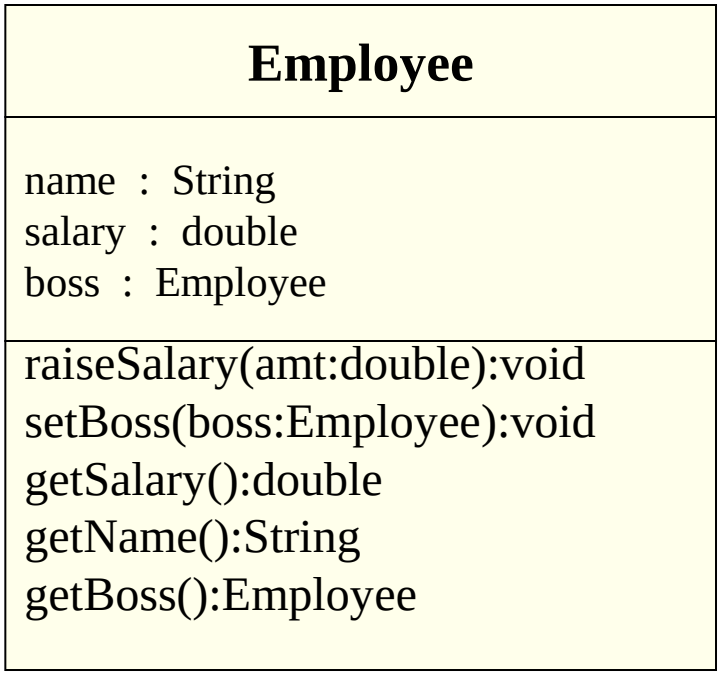
The kind of operations we will need include:

Increase Salary	raiseSalary(double)
Set the immediate boss	setBoss(Employee)
Get employee details	getName(), getBoss() …
Constructor to set name, salary	Employee(String,double)

Three employees *Bill* (big boss), *David* (small boss) and *Mike*

We could model these employees and their relationship using `Employee` class as shown below.

UML class diagram



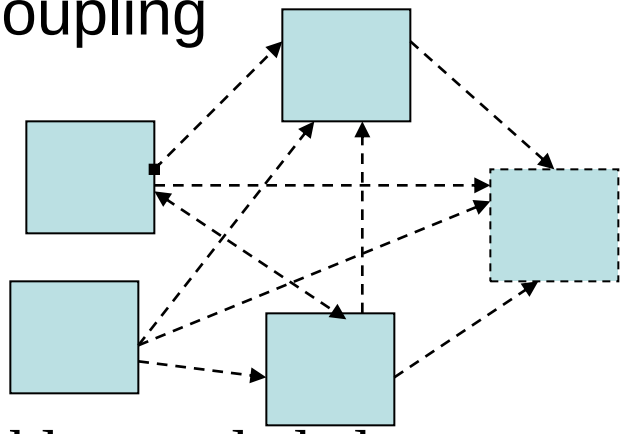
```
public class TestEmployee
{  public static void main(String args[])
   {  // constructing 3 Employee objects
      Employee bill = new Employee("Bill Gates",100000.0);
      Employee david = new Employee("David Paul",80000.0);
      Employee mike = new Employee("Mike Mogan",50000.0);
      david.setBoss(bill);
      mike.setBoss(david);
      mike.raiseSalary(10000);
      david.raiseSalary(20000);

      // printing details of mike
      System.out.print("name = " + mike.getName());
      System.out.print(" Salary = " + mike.getSalary());
      Employee boss = mike.getBoss();
      // if mike has a boss print his details
      if (boss != null)
         System.out.println("Reports to "+boss.getName());
      else System.out.println();
   }
}
```

```
public class Employee  {
   public Employee(String empName, double empSalary)
   {  name = empName;
      salary = empSalary;
   }
   private String name;
   private double salary;
   private Employee boss;

   public void raiseSalary(double amount) {
         salary = salary + amount;
   }

   public void setBoss(Employee empBoss) {
         boss = empBoss;
   }
```

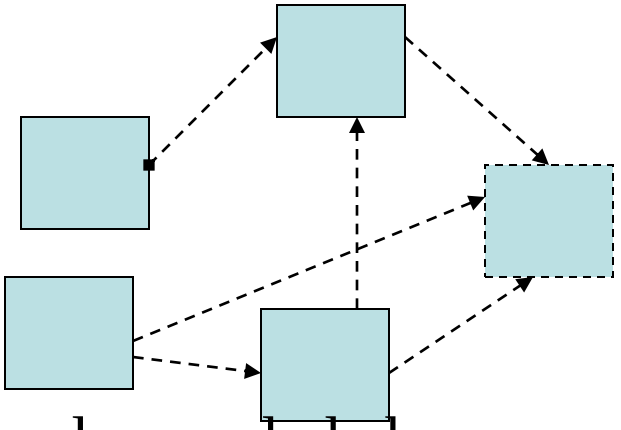
```
    public double getSalary() {
        return salary;
    }

    public String getName() {
        return name;
    }

    public Employee getBoss()   {
        return boss;
    }
}
```

Some guidelines for classes

- A class should represent single concept – methods must be cohesive (Cusomer, Account, Part, …)
- When a class has multiple concepts consider separating them.
- If a class A is dependent class B, changes to class B will affect also affect class A. Hence a good class design should aim to reduce dependencies or coupling



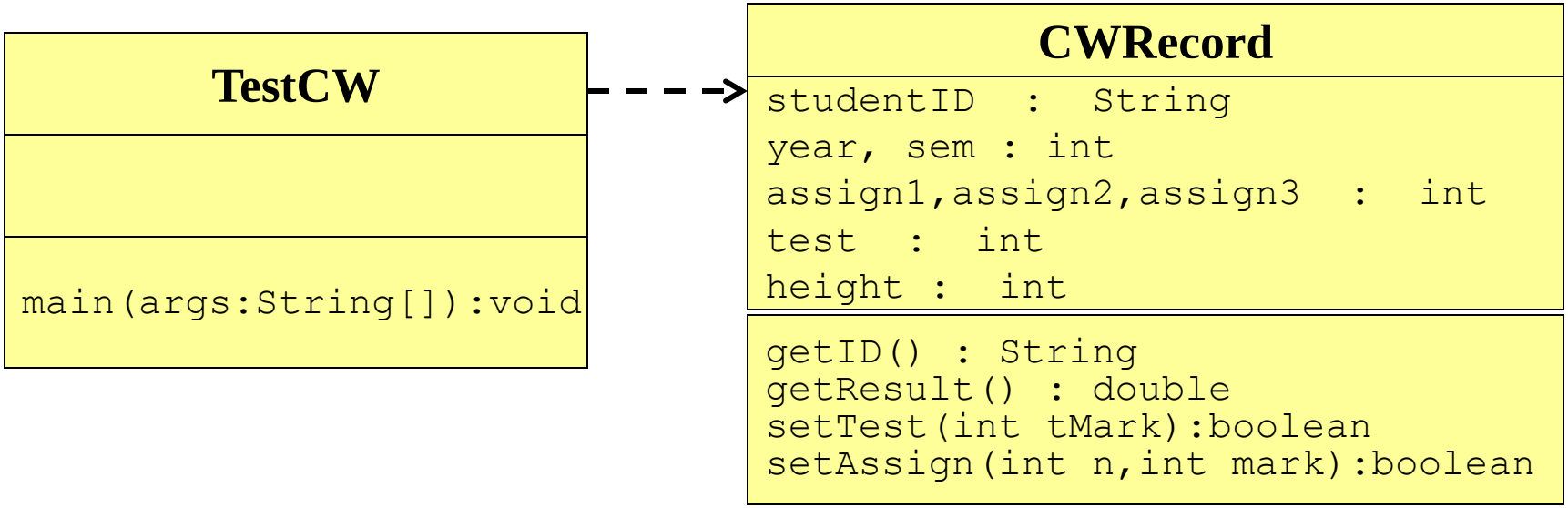
Highly coupled classes



Lowly coupled classes

Class Application -1

- You are required to complete the program to help keep track of student coursework made up of 3 components:
  - 3 assignments 20% each
  - 1 test 40%
- All components are marked out of 100
- Marks outside range 0-100 to be ignored
- class UML diagram specified.

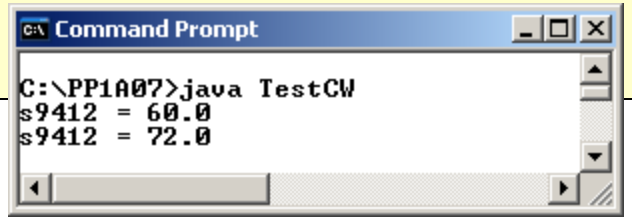


TestCW class and Expected results

```
public class TestCW
{   public static void main(String args[])
   {
      CWRecord mikeCW = new CWRecord("s9412",2007,1);
      mikeCW.setAssign(1,60);
      mikeCW.setAssign(2,80);
      mikeCW.setAssign(3,160);
      mikeCW.setTest(80);
      System.out.println(mikeCW.getID() + " = " +
                         mikeCW.getResult());
      mikeCW.setAssign(3,60);

      System.out.println(mikeCW.getID() + " = " +
                         mikeCW.getResult());
   }
}
```

Assignment 3 mark ignored (outside range)

Assignment 3 mark reset



class CWRecord

```
class CWRecord
{
   private String studentID;
   private int year;
   private int sem;
   private int assign1;
   private int assign2;
   private int assign3;
   private int test;

   public CWRecord(String ID, int yr, int sm)
   {
      …
      …
      …
   }
   public String getID()
   {
      …
   }
   public boolean setAssign(int n, int mark)
   {
      // ignore mark if outside range and return false
      // otherwise store mark in assign1,2 or 3 and return true
      …
      …
      …
      …

   }
```

class CWRecord cont'd

```
    public boolean setTest(int tMark)
    {
      // ignore tMark if outside range and return false
      // otherwise store tMark in mark and return true




    }
    public double getResult()
    {
      // compute and return overall mark based of formula given


    }
}
```

Class Application 2: Part class

Write a class named Part to model a motor vehicle parts bins (axles, radiators, …) maintained by a supplier. It should have instance variables to store the ID, name, stock-level, reorder-level and unit-price. It should provide a constructor taking as arguments ID, name, stock-level and reorder-level and unit-price (price per item).

It should provide accessors to get the ID, name, stock-level, reorder-level and unit-price. It should provide another method to print the part details (to System.out) including ID, name, stock-level, reorder-level and unit-price.

It should provide two mutators one to supply and another to replenish, both taking an integer (quantity) as argument. The replenish method should add the quantity to the stock level. The supply() method should deduct the stock-level by the specified quantity and return the cost, if sufficient stock is available. If the new stock level falls below the reorder level a message of the form “Reoroder stock with ID xxx” must be printed by this method. If insufficient stock available supply() method must return a -1.0.

Test all the method of the class (including constructor).

StringTokenizer class

•Many applications require reading more than one field from a line.

•Suppose, we have to sum all numbers until an empty line is input.

Please enter the numbers to be summed (Empty line to terminate)
8 5 9
7 8 9 5
6 7 4
←——— Terminate (Empty String)
Total = 68

- To extract one token at a time from a line we can use the StringTokenizer class.
- The constructor takes a String as argument. Constructor with one argument uses default delimiters (" `\t\n\r`") . Constructor with 2 arguments allow specific delimiter to be passed as second String argument
- Method `countTokens()` to find the number of tokens.
- Method `nextToken()` to get the next token

StringTokenizer Application

```
import java.util.*;
public class StringTokenizerDemo
{  public static void main(String args[])
   {  Scanner sc = new Scanner(System.in);
      System.out.println("Enter Numbers row by row : ");
      System.out.println("Empty line to terminate : ");
      int sum = 0;
      String s = sc.nextLine();
      while ( s.length()>0 )
      {
         StringTokenizer st=new StringTokenizer(s);
         int n = st.countTokens();
         for (int i=0; i<n; i++)
            sum += Integer.parseInt(st.nextToken());
         s = sc.nextLine();
      }
      System.out.println("Total = " + sum);
   }
}
```

Exercise StringTokenizer

Write a program using StringTokenizer to repeatedly read the names in the order First Name, Middle Name and Surname and to display them in reverse order. Terminate when an empty string is entered.

Please enter first, middle and surname (Empty line to terminate)
Tim Abraham Lincoln
Lincoln Abraham Tim
Mike Richard Cooper
Cooper Richard Mike
Terminate (Empty String)

StringTokenizer Exercise

Regular Expressions

- Regular expression is a string that describes a pattern for matching a set of strings.
- It can be used for matching replacing and splitting strings.
- It is more powerful and flexible than StringTokenizer for splitting

Matching Strings

- matches() allows a String to be matched against a fixed string or a pattern.
- The following code segment prints matches.

```
if ("Java".matches("Java"))
    System.out.println("matches");
```

- The following code segments prints `03-95671234 is a Victorian phone number`

```
String phone = "03-95671234“;
String pattern1 = "03-\\d{8}";
if ( phone.matches(pattern1) )
     System.out.println(s1+" is a Victorian phone number");
```

Regular Expression Syntax

- . Matches any character ”Computer”.matches(”C……r”)
- (pq|rs) matches pq or rs ”Dog”.matches(”D(at|og)”)
- [aeiou] matches a, e, i, o, u ”This”.matches(”Th[aeiou]s”)
- [^aeiou] matches characters others a, e, i, o, u ”This”.matches(”T[^aeiou]is”)
- [a-d] matches a through d ”ball”.matches(”[a-d]all”)
- \d matches any digit “Java2".matches(“Java\\d")
- \D matches any non-digit “Java".matches(“Jav\\D")
- \w matches any word character (alphanumeric) “ab".matches(“\\wb")
- \W matches any non-word char “$b".matches(“\\Wb")

Regular Expression Syntax cont'd

- **\s matches any whitespace char "ab c".matches("ab\\wc")**
- **\S matches any non whitespace char "abc".matches("a\\Wc")**
- **p* matches 0 or more occurence of pattern p "abcd".matches("a(bc)*bcd")**
- **p+ matches 1 or more occurence of pattern p "abcd".matches("a(bc)+d")**
- **p+ matches 1 or more occurence of pattern p "abcd".matches("a(bc)+bcd")**
- **P{n} matches n occurrence of pattern p "abcbcd".matches("a(bc){2}d")**
- **P{m,n} matches between m and n occurrence of pattern p "abcbcd".matches("a(bc){1,3}d")**

Take Note

- Backslash is a special character hence the need to uoe \\d, \\w, \\s etc
- \\s are characters tab, space, newline, carriage return
- \\w are same as [a-zA-Z0-9_]
- Use () to group patterns (xy){2} matches xyxy but xy{2}matches xyy.

Regular Expression : Application

Write a program to read a the student number and assignment marks (up to 5 separated by single tab) and find the overall mark. The program should terminate when user enter an empty string.

Sample Input/Output

Enter Student ID and marks (empty string to terminate)

S123 4 8 9

S123 total = 21

S124 9 7 3 6

S124 total = 25

Terminate (Empty String)

Using split and Regular Expressions

```
import java.util.*;
public class Tokenize
{  public static void main(String args[])
   {
      System.out.println("Enter ID & marks separate by tabs");
      System.out.println("Enter empty line to terminate");
      Scanner console = new Scanner(System.in);
      String s, snum;
      s = console.nextLine();
      while (s.length() != 0)
      {
         String tokens[] = s.split("\\s");
         snum = tokens[0];
         …
         …

         s = console.nextLine();
      }
   }
}
```